Comparison of Two Methods for Computing Action Values in XCS with Code-Fragment Actions

Muhammad Iqbal
Victoria University of Wellington, NZ
muhammad.iqbal@

Will N. Browne
Victoria University of Wellington, NZ
will.browne@

Mengjie Zhang
Victoria University of Wellington, NZ
mengjie.zhang@ecs.vuw.ac.nz

ABSTRACT

XCS is a learning classifier system that uses accuracy-based fitness to learn a problem. Commonly, a classifier rule in XCS is encoded using a ternary alphabet based condition and a numeric action. Previously, we implemented a code-fragment action based XCS, called XCSCFA, where the typically used numeric action was replaced by a genetic programming like tree-expression. In XCSCFA, the action value in a classifier was computed by loading the terminal symbols in the action-tree with the corresponding binary values in the condition of the classifier rule. This enabled accurate, general and compact rule sets to be simply produced. The main contribution of this work is to investigate an intuitive way, i.e. using the environmental instance, to compute the action value in XCSCFA, instead of the condition of the classifier rule. The methods will be compared in five different Boolean problem domains, i.e. multiplexer, even-parity, majority-on, design verification, and carry problems. The environmental instance based XCSCFA approach had better classification performance than standard XCS as well as classifier condition based XCSCFA and solved all the problems experimented here. In addition it produced more general and compact classifier rules in the final solution. However, classifier condition based XCSCFA has the advantage of producing the optimal classifiers such that they are clearly separated from the sub-optimal ones in certain domains.

Categories and Subject Descriptors

F.1.1 [**Models of Computation**]: Genetics-Based Machine Learning, Learning Classifier Systems

General Terms

Algorithms, Performance

Keywords

Learning Classifier Systems, XCS, XCSCFA, Code-Fragments, Boolean Problems, Pattern Recognition

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'13 Companion, July 6–10, 2013, Amsterdam, The Netherlands.
Copyright 2013 ACM 978-1-4503-1964-5/13/07 ...$15.00.

1. INTRODUCTION

A learning classifier system (LCS) is a rule-based online learning system that adaptively learns a task by interacting with an unknown environment and uses evolutionary computing to evolve the rules according to the reinforcement received from the environment. XCS [24] is a formulation of LCS that uses accuracy-based fitness to learn the problem. Each rule in XCS is of the form 'if *condition* then *action*', having two parts: a condition and the corresponding action. Commonly, the condition is represented by a fixed length bitstring defined over the ternary alphabet $\{0, 1, \#\}$ where '#' is the 'don't care' symbol which can be either 0 or 1; and the action is represented by a numeric constant.

Previously, we implemented a code-fragment action based XCS, called XCSCFA [13], in order to evolve optimum populations; where the typically used numeric action was replaced by a tree-expression similar to a tree generated in genetic programming [19]. In XCSCFA, the action value in a classifier was computed by loading the terminal symbols in the code-fragment action with the corresponding binary values in the condition of the classifier rule. A '#' symbol in the condition of the classifier rule was randomly treated as 0 or 1, resulting in inconsistent actions. The XCSCFA system has outperformed standard XCS in various Boolean problem domains [15].

In the work presented here, the action value of a classifier rule in XCSCFA will be computed using the environmental state, instead of the condition of the classifier rule. The results will be tested and compared in five different Boolean problem domains, i.e. multiplexer, even-parity, majority-on, design verification, and carry problems. These are complex problem domains having overlapping, niche imbalance, and epistatic properties (see section 4).

2. BACKGROUND

2.1 Learning Classifier Systems

Traditionally, an LCS represents a rule-based agent that incorporates evolutionary computing and machine learning to solve a given task by interacting with a previously unknown environment. After observing the current state of the environment, the agent performs an action and the environment provides a reward. The generalization property in LCS allows a single rule to cover more than one state provided that the action-reward mapping is similar. Traditionally, generalization in LCS is achieved by the use of the special 'don't care' symbol '#' in classifier conditions, which

matches any value of a specified attribute in the vector describing the current environmental state.

XCS is a formulation of LCS that uses accuracy-based fitness to learn the problem by forming a complete mapping of states and actions to rewards. In XCS, the learning agent evolves a population $[P]$ of classifiers, where each classifier consists of a rule and a set of associated parameters estimating the quality of the rule. Each classifier has three main parameters: 1) prediction p, an estimate of the payoff expected from the environment if its action is executed; 2) prediction error ϵ, an estimate of the errors between the predicted payoff and the actually received reward; and 3) fitness F, an estimate of the classifier's utility. In addition, each classifier keeps an experience parameter exp, which is a count of the number of times it has been updated, and a numerosity parameter n, which is a count of the number of copies of each unique classifier.

XCS operates in two modes, explore and exploit. In the explore mode, the agent attempts to obtain information about the environment and describes it by creating the decision rules, using the following steps:

- observes the current state s of the environment, usually represented by a fixed length bitstring defined over the binary alphabet $\{0, 1\}$.
- selects classifiers from the classifier population $[P]$ that have conditions matching the state s, to form the match set $[M]$.
- performs covering: for every action $a_i \in A$ in the set of all possible actions, if a_i is not represented in $[M]$ then a random classifier is generated with a given generalization probability such that it matches s and advocates a_i, and added to the set $[M]$ as well as to the population $[P]$.
- forms a system prediction array, $P(a_i)$ for every $a_i \in A$ that represents the system's best estimate of the payoff should the action a_i be performed in the state s.
- selects an action a to explore (probabilistically or randomly) and selects all the classifiers in $[M]$ that advocated a to form the action set $[A]$.
- performs the action a, records the reward r received from the environment, and uses r to update the associated parameters of all classifiers in $[A]$.
- when appropriate, implements rule discovery by applying an evolutionary mechanism (commonly a genetic algorithm) in the action set $[A]$, to introduce new classifiers to the population.

Additionally, the explore mode may perform subsumption deletion to merge specific classifiers into any more general and accurate classifiers. Subsumption deletion is a way of biasing the genetic search towards more general, but still accurate, classifiers [3]. It also effectively reduces the number of classifier rules in the final population [18]. In contrast to the explore mode, in the exploit mode the agent does not attempt to discover new information and simply performs the action with the best predicted payoff. The exploit mode is also used to test learning performance of the agent.

Various richer encoding schemes have been investigated to represent high level knowledge in LCS in an attempt to obtain compact classifier rules [2, 7, 12], to reach the optimal performance faster [20, 22], to approximate functions [4, 25], to learn problems involving a large number of actions [21], to develop useful feature extractors [1], and to identify and process building blocks of knowledge [5, 16].

2.2 Previous Work on Code-Fragment Classifier Systems

The main goal of the research direction is to develop a scalable classifier system. To achieve this goal, we implemented a GP-like rich encoding scheme, in the conditions of classifier rules, in an XCS based system to identify building blocks of knowledge [17]. Then, the fitter building blocks of information were extracted, in the form of tree-like code fragments, from low dimensional problems and reused in learning more complex high scale problems in the domain [11, 16].

Subsequently, we implemented the code-fragment encoding scheme in action of a classifier rule in XCSCFA to evolve optimum populations [13]. The XCSCFA system outperformed standard XCS in various Boolean problem domains [15]. We also implemented code-fragment based action in real-valued XCSRCFA to compute continuous action [12]. Recently, we implemented a state machine based XCS classifier system, known as XCSSMA [14], where the action in a classifier rule was replaced by a Moore state machine [23]. XCSSMA produced the optimal classifiers in the final solution set to provide general scalable solutions to solve any n-bits even-parity problems and any n+n bits carry problems.

3. XCS WITH CODE-FRAGMENT ACTION

In XCSCFA, the typically used numeric action in a classifier rule is replaced by a GP-tree like code fragment. Each code fragment is a binary tree having maximum seven nodes. The function set for the action trees is {AND, OR, NOT, NAND, NOR} and the terminal set is {D0, D1, D2, ... Dn-1}, where n is the length of an environmental instance. A population of classifiers having code-fragment actions is illustrated in Table 1. The symbols &, |, ~, d, and r denotes AND, OR, NOT, NAND, and NOR operators respectively. The code-fragment trees are shown in postfix form.

Table 1: A population of classifiers in XCSCFA.

No.	Condition						Action
1	0	1	#	1	#	0	D5 D5 d
2	1	0	0	0	1	#	D4
3	0	1	#	1	#	1	D3 D0 | ~
4	0	0	0	#	#	#	D1 D4 r
5	1	1	#	#	0	0	D1 D4 &
...	...						...

In XCSCFA, the action value is computed by loading the terminal symbols in the action tree with corresponding binary values from the condition in the classifier rule, and a 'don't care' symbol in the condition is randomly treated as 0 or 1. For example, consider the classifier rule shown in Fig. 1 and the environmental input message '110010'. In the condition of this classifier rule, D2 is a '#' symbol. To compute the action value in this classifier rule, D2 will be loaded with 0 or 1 randomly. Now, if the binary value taken for D2 is 0 then the action will be 0, otherwise 1. Hence, the action value in XCSCFA may vary, even for the same environmental instance, at different times during the training process; unlike standard XCS.

XCS keeps a complete map, i.e. the classifiers advocating consistently correct classification (and hence predicting accurate environmental reward of say 1000) as well as the classifiers advocating consistently incorrect classification (and hence predicting accurate environmental reward of say 0).

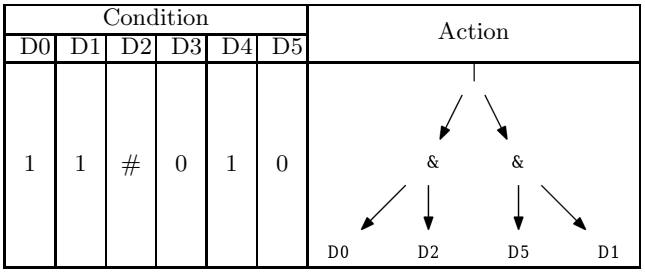

Figure 1: A classifier with code-fragment action.

It is noted that the building blocks of information in the condition of an accurate incorrect classifier are exactly the same as in the counterpart correct classifier. For example, '000### : $1 \rightarrow 0$' is an accurate incorrect classifier which has the same condition as that in the counterpart accurate correct classifier '000### : $0 \rightarrow 1000$'. The rule discovery operation is applied in the action set, which is formed by the classifiers advocating a *certain* action, commonly selected at random, and covering the currently observed environmental input. As all the classifiers in an action set advocate the same action, the correct and incorrect classifiers cannot occur in the same action set, so cannot be simultaneously used in breeding of the new classifiers. This means, in an XCS system that although both correct and incorrect classifiers are kept throughout the learning of the system, the building blocks of information in them are not efficiently exploited as they are not allowed to take part in the same breeding operation. The XCS ability to keep a complete map combined with the inconsistent actions may preserve important building blocks of information in XCSCFA. Due to inconsistent action values, the incorrect classifiers can occur in the action set as correct classifiers in XCSCFA so can be used for the production of good classifiers. For details of the rule discovery operation in XCSCFA, the interested reader is referred to the original XCSCFA paper [13].

In this work, a more intuitive implementation of XCSCFA is presented where the action value is computed by loading the terminal symbols with the corresponding binary values from the environmental input instead of the classifier condition. For sake of readability the two implementations of XCSCFA are named XCSCFA$_c$ (i.e. classifier based) and XCSCFA$_e$ (i.e. environment based). It is to be noted that a classifier rule in XCSCFA$_e$ can have different action values for different environmental instances, but for the same instance it will have the same action value at any time during the training process. Previously in XCSCFA$_c$ [13], it was necessary for a subsumer classifier to have a consistent action value as a precaution to avoid any volatility in the performance of the classifier system. In the work presented here this restriction has been removed in order to perform fair comparison with XCS; and to determine whether the XCSCFA$_c$ and XCSCFA$_e$ systems can produce more compact final solutions.

4. EXPERIMENTAL DESIGN

4.1 The Problem Domains

The problem domains used in the experimentation are the multiplexer, even-parity, majority-on, design verification, and carry problems.

A multiplexer is an electronic circuit that accepts input strings of length $n = k + 2^k$, and gives one output. The values of the k so-called address bits are used to select one of the 2^k remaining data bits to be given as output. For example in 6-bits multiplexer, if the input is 011101 then the output will be 1 as the first two bits 01 represent the index 1 (in base ten), which is the second bit following the address. Multiplexer problems are highly non-linear and have epistasis, i.e. importance of data bits is dependent on address bits.

In even-parity problems, the output depends on the number of ones in the input instance. If the number of ones is even, the output will be one, and zero otherwise. Using the ternary alphabet based conditions with the static numeric action, no useful generalizations can be made for the even-parity problems.

The majority-on problems are similar to even-parity problems in that the output depends on the number of ones in the input instance. If the number of ones is greater than the number of zeros, the problem instance is of class one, otherwise class zero. In majority-on problem domain, the complete solution consists of strongly overlapping classifiers, so is therefore difficult to learn. For example, '1##11:1' and '11#1#:1' are two maximally general and accurate classifiers, but they overlap in the "11*11" subspace.[1]

Digital design verification is a real world problem domain where a digital design is verified, to discover bugs, before manufacturing the actual system. The design verification problem experimented here is a 7-bits Boolean example of a simulation-based DV problem, known as DV1, introduced by Ioannides et al. in [10]. The Sigma notation [8] is a compact representation to denote a Boolean function by listing each onset row from the truth table of the function. For example, the function $\overline{x}y + x\overline{y}$ can be denoted by the Sigma notation $\Sigma(1, 2)$. The DV1 problem in the Sigma notation is represented as Σ(1, 2, 3, 8, 9, 10, 11, 13, 14, 24, 25, 26, 27, 28, 30, 40, 41, 42, 43, 46, 47, 56, 57, 58, 59, 61, 65, 66, 67, 69, 70, 71, 72, 73, 74, 75, 77, 78, 79, 81, 82, 83, 85, 86, 88, 89, 90, 91, 93, 94, 95, 97, 98, 99, 101, 102, 103, 104, 105, 106, 107, 109, 110, 113, 114, 115, 117, 118, 121, 122, 123, 125, 126, 127). Similar to majority-on problems, the complete solution for the DV1 problem consists of overlapping classifiers.

In the carry problem, two binary numbers of the same length are added. If the result triggers a carry, then the output is one otherwise zero. For example, in case of three bits numbers 101 and 010, the output is 0, whereas for the numbers 110 and 100 the output is 1. Similar to majority-on and DV1 problems, the complete solution in carry problem domain consists of overlapping classifiers, and in addition it is a niche imbalance problem domain.

4.2 Experimental Setup

The system uses the following parameter values, commonly used in the literature, as suggested by Butz and Wilson in [6]: learning rate $\beta = 0.2$; fitness fall-off rate $\alpha = 0.1$; prediction error threshold $\epsilon_0 = 10$; fitness exponent $\nu = 5$; threshold for GA application in the action set $\theta_{GA} = 25$; two-point crossover with probability $\chi = 0.8$; mutation probability $\mu = 0.04$; experience threshold for classifier deletion $\theta_{del} = 20$; fraction of mean fitness for deletion $\delta = 0.1$; classifier experience threshold for subsumption

[1]Here, * can be 0, 1, or #.

$\theta_{sub} = 20$; probability of 'don't care' symbol in covering $P_{\#} = 0.33$; reduction of the fitness $fitnessReduction = 0.1$; and the selection method is tournament selection with tournament size ratio 0.4. Both GA subsumption and action set subsumption are activated. The number of classifiers used is 2000 and the number of training examples is half a million, in all the experiments conducted here. Explore and exploit problem instances are alternated. The reward scheme used is 1000 for a correct classification and 0 otherwise. All the experiments have been repeated 30 times with a known different seed in each run. Each result reported in this work is average of the 30 runs.

In all graphs presented here, the X-axis is the number of problem instances used as training examples, the Y-axis is the classification performance measured as the moving average over the last 1000 exploit problem instances, and the error bars show standard deviation in the 30 runs.

5. RESULTS

5.1 The Multiplexer Problem Domain

The results of standard XCS and the two code-fragment action based XCSCFA methods for the 20-bits multiplexer problem are shown in Fig. 2. All of the three methods successfully solved the problem, but XCSCFA$_c$ took more instances to reach a similar performance level.

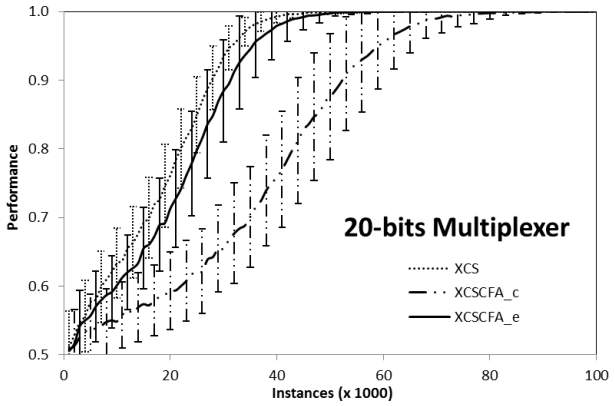

Figure 2: Results of 20-bits multiplexer problem.

The multiplexer is a niche balanced problem domain and there exists a complete solution that does not contain any overlapping classifier rules. Hence, the standard XCS effectively solved the 20-bits multiplexer problem. The XCSCFA methods used more training examples due to the increased search space.

5.2 The Even-Parity Problem Domain

The performance of standard XCS and the XCSCFA methods in learning 7-bits even-parity problem is shown in Fig. 3. It is observed that standard XCS and XCSCFA$_c$ could not learn the 7-bits even-parity problem, whereas XCSCFA$_e$ has successfully solved it.

The even-parity domain does not allow generalizations if the standard ternary alphabet based encoding scheme is used with static numeric action. So each bit must be specific for a rule to be accurate, requiring 2^8 such rules for the 7-bits even-parity problem. The standard XCS and XCSCFA$_c$ methods were not able to evolve enough accurate

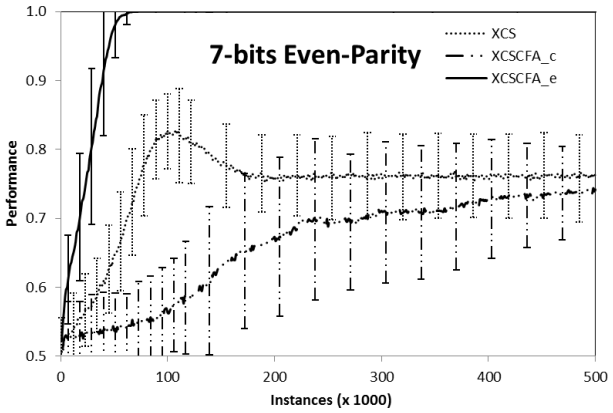

Figure 3: Results of 7-bits even-parity problem.

rules, given the experimental setup in section 4.2. However, the XCSCFA$_e$ method solved the 7-bits even-parity problem by producing generalized classifier rules, see section 6.

5.3 The Majority-on Problem Domain

The complete solution of the majority-on problem domain consists of strongly overlapping classifiers in the final solution, which makes it a hard problem to learn. The performance of standard XCS and the XCSCFA methods in learning 5-bits majority-on problem is shown in Fig. 4. It is observed that standard XCS reached approximately 94% performance, but could not completely solve the 5-bits majority-on problem, whereas the XCSCFA methods have solved it using approximately $10,000$ training examples.

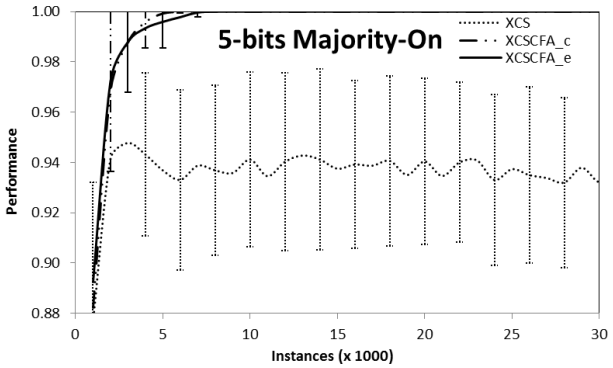

Figure 4: Results of 5-bits majority-on problem.

5.4 The Design Verification Problem Domain

The performance of standard XCS and the XCSCFA methods in learning the DV1 problem is shown in Fig. 5. It is observed that standard XCS achieved approximately 97% performance level, but could not completely solve the DV1 problem, whereas the XCSCFA methods have successfully solved it. Ioannides et al. [9] improved the performance of XCS in the DV1 problem to 99.76%, by modifying the standard fitness update procedure and using an individually computed learning rate for each classifier, but still could not completely solve it.

5.5 The Carry Problem Domain

The complete solution in the carry problem domain consists of overlapping classifiers, in addition it is a niche im-

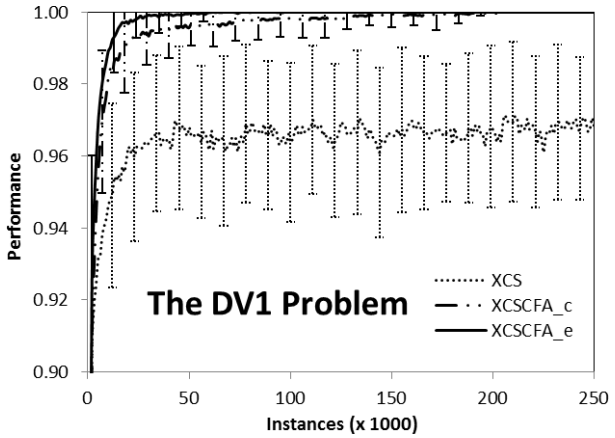

Figure 5: Results of the DV1 problem.

balance domain, which makes it very difficult to learn. The performance of standard XCS and the XCSCFA methods in learning 3+3 bits carry problem is shown in Fig. 6. It is observed that standard XCS reached approximately 97% performance, but could not completely solve the 3+3 bits carry problem, whereas the XCSCFA methods have successfully solved it.

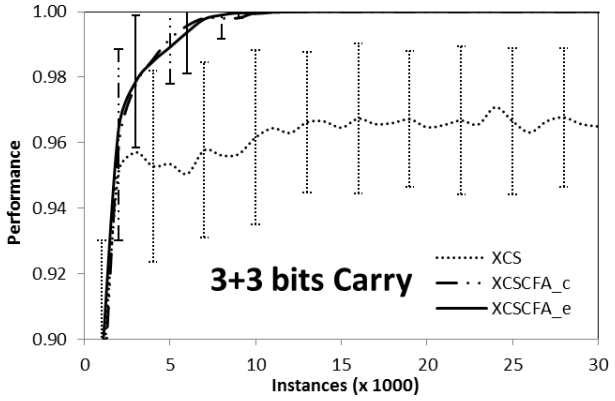

Figure 6: Results of 3+3 bits carry problem.

6. ANALYSIS OF EVOLVED CLASSIFIERS

6.1 The Multiplexer Problem Domain

Every optimal classifier, encoded in ternary representation, for the 20-bits multiplexer problem contains five specific bits in the condition. Four of these are the address bits and the fifth is corresponding data bit, see section 4.1, e.g. '0101#####1########## : 1'. The solution obtained using XCS contains both optimal and sub-optimal classifier rules, see Table 2, that needs extensive post-processing to separate these rules [18]. The XCSCFA$_c$ system has the advantage of producing the optimal classifiers separated from the sub-optimal ones with respect to the numerosity values, see Table 3 and refer to [13] for more examples and details.

The XCSCFA$_e$ method produced more general and compact classifiers than the other two methods, see Table 4. These may appear over-general as they have more '#' symbols than the optimal ternary representation classifiers. However, they are perfectly accurate for all the matching environmental instances due to the referencing of specific bits in the code-fragment action. For example, the classifier number 10 '1010############### : D14' does not contain any specific data bit in the condition, but the addressed data bit is in the action, i.e. D14. Now, the action value of this classifier is 1 if the environmental bit D14 is 1 and 0 otherwise. Therefore, it is correct for all the matching instances. It is to be noted that the compact optimal classifiers produced in XCSCFA$_e$ are not clearly separated from the sub-optimal ones; unlike XCSCFA$_c$.

The XCSCFA systems have larger final populations than XCS due to the multiple genotypes to a single phenotype mapping in the code-fragment action trees. As there are multiple classifiers in the XCSCFA systems for a corresponding single classifier in standard XCS, the fitness values of the final classifiers are relatively smaller in the former systems.

6.2 The Majority-On Problem Domain

The complete solution in the majority-on problem domain consists of overlapping classifiers, so is hard to learn. It is observed that standard XCS produced overgeneral classifiers in the 5-bits majority-on problem, see Table 5. For example, the first classifier rule '#0#0# : 0' is an overgeneral classifier which matches eight environmental instances: '00000', '00001', '00100', '00101', '10000', '10001', '10100', and '10101'. It is a "dangerous" classifier because it is correct for all the matching instances except '10101', so likely to be considered accurate ($\epsilon < \epsilon_0$) and therefore, it is highly likely that it will subsume the specific but correct classifier '10101 : 0' in the training process.

There are a few maximally general and accurate classifiers in the final solution, and they have relatively low experience values, e.g. the classifiers number 11 to 15 in the Table 5. These low experience values indicate that the overgeneral classifiers like number 1 and 9, having $\epsilon < \epsilon_0$, would have subsumed the otherwise needed accurate classifiers, resulting in poor performance of the system.

The XCSCFA systems managed to avoid overgeneral classifier rules in the final solutions, see Table 6 and Table 7. The XCSCFA$_e$ method produced more general (and still accurate) classifiers than the other two methods, similar to the multiplexer domain. For example, the classifier number 9 '1##1# : D2 D1 D4 | |' matches eight environmental instances '10010', '10011', '10110', '10111', '11010', '11011', '11110', and '11111'. Now, the action value of this classifier is 1 if any of the environmental bits D1, D2, or D4 is 1. Therefore, it is correct for all the matching instances.

6.3 The Even-Parity Problem Domain

In standard XCS with numeric action, it is not possible for a classifier rule to have a '#' symbol in the condition and still be accurate for the even parity problems. However, it is interesting to note that XCSCFA$_e$ was able to produce accurate general classifier rules having a '#' symbol in the condition for the 7-bits even parity problem, see Table. 8. For example, the second classifier rule in Table. 8 is '101010# : D6' which matches two environmental instances, '1010100' and '1010101'; and the action value is the binary value of the last symbol in the environmental instance, i.e. D6. Therefore the advocated action for the problem instance '1010100' is 0 and for '1010101' is 1, and both are accurate. It is to be noted that in the 7-bits even parity problem, the classifiers

Table 2: A sample of classifiers from final solution obtained in XCS for the 20-bits multiplexer problem.

No.	Condition	Action	n	F	ϵ	p	exp
1	0010##0############	0	24	0.81	0	1000	455
2	00001##############	0	23	0.92	0	0	7281
3	0010##0############	1	22	0.79	0	0	7007
4	0101#####1#########	1	19	0.62	0	1000	7542
5	0111#######0########	1	18	0.66	0	0	7358
6	0110######1#########	1	17	0.69	0	1000	7324
7	0111#######1########	0	15	0.53	0	0	6013
8	1010##########1#####	1	14	0.64	0	1000	3841
9	00000##############	0	11	0.54	0	1000	74
10	00#01#1############	1	8	0.22	0	1000	1131
11	011#######00########	1	8	0.23	0	0	1810
12	11#0############0#0#	0	8	0.27	0	1000	2483
13	0#000###0##########	0	8	0.29	0	1000	6435
14	00#1#1#1##########	0	8	0.23	0	0	7170
15	000#00############	1	7	0.21	0	0	193

Table 3: A sample of classifiers from final solution obtained in XCSCFA$_c$ for the 20-bits multiplexer problem.

No.	Condition	Code-Fragment Action	n	F	ϵ	p	exp
1	1011##########1####	D2 D0 & D0 &	9	0.24	0	1000	586
2	0010##0############	D6	9	0.28	0	1000	1076
3	1100##########1###	D1 D13 | ~	9	0.35	0	0	1455
4	1001########0######	D13 D9 d D13 D9 d d	9	0.29	0	1000	2080
5	0111#######1########	D0 D0 d D0 D0 d r	9	0.23	0	0	4894
6	1000########1#######	D2 D2 d D2 d	8	0.2	0	1000	156
7	1000########0#######	D6 D0 ~ d	8	0.24	0	0	1563
8	0111#######0########	D11 D11 D13 & r	8	0.25	0	0	2226
9	0101#####0#########	D11 D0 & D11 D0 & &	8	0.36	0	1000	2494
10	1100##########1###	D3 D2 d ~	8	0.31	0	0	2445
11	1111################1	D3	8	0.27	0	1000	4295
12	1011##########1####	D0 ~ ~	8	0.21	0	1000	5468
13	1100##########1###	D11 D16 | ~	8	0.32	0	0	5657
14	1011##########0####	D15 D15 d	7	0.23	0	0	358
15	1010##########1#####	D14 D1 r D1 r	7	0.21	0	1000	405

Table 4: A sample of classifiers from final solution obtained in XCSCFA$_e$ for the 20-bits multiplexer problem.

No.	Condition	Code-Fragment Action	n	F	ϵ	p	exp
1	010#####0##########	D9 D3 & ~	12	0.22	0	0	1968
2	1110################	D0 D18 d	12	0.25	0	0	3659
3	1110################	D18 ~	12	0.22	0	0	13580
4	1000################	D2 D12 r	11	0.19	0	0	5094
5	1000################	D12 D0 D0 d r	11	0.2	0	0	12093
6	0010################	D2 D6 d D2 D6 d r	10	0.16	0	1000	1944
7	1111################	D3 D1 9 D2 & d	10	0.16	0	0	5939
8	1100################	D16 D2 r	9	0.19	0	0	1859
9	1011################	D15 ~ ~	9	0.18	0	1000	4804
10	1010################	D14	9	0.16	0	1000	5660
11	1011################	D15 ~	9	0.2	0	0	5843
12	0100####0##########	D2 ~ D2 ~ |	9	0.19	0	0	3916
13	0010################	D6 ~	9	0.16	0	0	7796
14	0000################	D4 ~ ~	9	0.19	0	1000	10549
15	101############1####	D14 D3 r ~	8	0.1	0	1000	1882

obtained using XCSCFA$_e$ have sufficiently higher fitness values than other domains, because it is relatively difficult to produce multiple classifiers covering the same niche. This is unlikely in other problem domains such as the multiplexer and majority-on problems.

Similar to standard XCS, the XCSCFA$_c$ method could not produce accurate general classifier rules in the 7-bits even parity problem, and failed to solve it with this setup. As a result of the generalized classifiers in XCSCFA$_e$, the number of classifiers required in the final solution set reduced to half of the number of specific classifiers needed otherwise and it successfully solved the 7-bits even parity problem. It is to be noted that there is just one '#' symbol in the condition of a general classifier produced in the XCSCFA$_e$ approach for the 7-bits even-parity problem. In XCSCFA$_c$ the classifiers having just one '#' symbol cannot be consistently accurate in the even-parity domain because the '#' symbol is randomly treated 0 or 1 in XCSCFA$_c$ to compute the action value of a classifier.

Table 5: A sample of classifiers from final solution obtained in XCS for the 5-bits majority-on problem.

No.	Condition	Action	n	F	ϵ	p	exp
1	#0#0#	0	108	0.93	0.09	999.99	18453
2	1#1##	1	93	0.7	203.53	799.51	238
3	##1#1	1	88	0.1	155.47	958.68	1414
4	#0#0#	1	83	0.37	23.34	4.09	1955
5	11###	0	74	0.15	136.84	46.46	2103
6	0#0##	0	66	0.45	11.34	998.47	4883
7	##11#	0	64	0.02	181.66	71.84	18909
8	0##0#	1	57	0.47	193.97	128.22	2164
9	##0#0	1	57	0.71	4.67	0.54	2104
10	##00#	0	50	0.01	106.5	973.67	7352
11	1#11#	0	22	0.93	0	0	66
12	#00#0	0	15	0.58	0	1000	62
13	11#10	1	3	0.53	0.72	1000	23
14	111##	0	1	0.36	5.15	0	3
15	0#00#	0	1	0.13	4.47	1000	4

Table 6: A sample of classifiers from final solution obtained in XCSCFA$_c$ for the 5-bits majority-on problem.

No.	Condition	Code-Fragment Action	n	F	ϵ	p	exp
1	00111	D1 D0 r	10	0.21	0	1000	3554
2	00101	D2 D4 & D4 &	7	0.14	0	0	1421
3	01001	D0 D1 D4 d r	7	0.14	0	0	1325
4	10101	D3 D3 & D1 |	7	0.14	0	0	1593
5	00101	D2 D4 d D2 D3 r |	7	0.14	0	1000	1703
6	01010	D3 D1 d D3 D1 d |	7	0.16	0	1000	1896
7	0#00#	D1 D0 &	6	0.11	0	1000	139
8	100#0	D0 D0 d	6	0.11	0	1000	70
9	000##	D2 D0 & D2 D0 & r	6	0.11	0	0	247
10	#1000	D4 D1 D2 r r	6	0.13	0	0	204
11	#1000	D1 D4 r	6	0.1	0	1000	376
12	1#000	D0 ~ D3 |	6	0.1	0	1000	429
13	01010	D3 D1 d D3 D3 d |	6	0.12	0	1000	234
14	1#11#	D0	6	0.11	0	1000	923
15	11#10	D3 D4 & D3 D4 & |	6	0.09	0	0	467

Table 7: A sample of classifiers from final solution obtained in XCSCFA$_e$ for the 5-bits majority-on problem.

No.	Condition	Code-Fragment Action	n	F	ϵ	p	exp
1	00#1#	D2 D4 d D2 D4 d |	15	0.13	0	0	3159
2	1##1#	D1 D4 | D2 D2 | r	13	0.12	0	0	10996
3	1#01#	D1 D4 | D1 D4 | d	13	0.12	0	0	16004
4	011##	D3 D4 r	13	0.12	0	0	17314
5	#1#10	D2 D0 r D3 &	13	0.11	0	0	19667
6	###00	D1 D2 D0 & d	11	0.1	0	0	6641
7	101#1	D1 D4 ~ |	11	0.11	0	0	2382
8	#1#00	D2 D0 & D2 D0 & d	11	0.1	0	0	5579
9	1##1#	D2 D1 D4 | |	11	0.1	0	1000	14545
10	0#01#	D1 D4 d ~	11	0.09	0	1000	11127
11	#1#10	D2 D0 | D2 D0 | |	10	0.09	0	1000	2927
12	##100	D1 D0 &	10	0.08	0	1000	6826
13	#01#0	D3 D0 & D0 &	10	0.09	0	1000	11246
14	0#0#1	D1 D3 d ~	9	0.08	0	1000	1017
15	#011#	D0 D4 | D0 D4 | |	9	0.07	0	1000	2093

7. CONCLUSIONS

The main aim of this work was to investigate the environmental instance based action value computation in XCSCFA. The environmental instance-based XCSCFA$_e$ system successfully solved all the experimented problems from five different complex Boolean domains whereas the classifier's condition-based XCSCFA$_c$ and the standard XCS systems failed in learning one and four of the five problems respectively. The XCSCFA$_e$ system produced more generalized classifiers, but the XCSCFA$_c$ system has the advantage of producing the optimal classifiers separated from the sub-optimal ones in certain domains.

It is anticipated that the XCSCFA systems outperformed standard XCS due to the following two properties of code-fragment based actions: 1) the ability in all accuracy-based systems to keep a complete map combined with inconsistent actions in XCSCFA systems preserve important building blocks of information, and 2) multiple genotypes for a single phenotype action provide diversity and redundancy, resulting in the robust XCSCFA systems especially in the

Table 8: A sample of classifiers from final solution obtained in XCSCFA$_e$ for the 7-bits even parity problem.

No.	Condition	Code-Fragment Action	n	F	ϵ	p	exp
1	10#1011	D2	22	0.99	0	0	2097
2	101010#	D6	19	0.97	0	1000	1092
3	1011#10	D1 D4 r	17	1	0	1000	576
4	000011#	D6	17	0.98	0	0	2016
5	1#01000	D1 D1 | D1 D1 | d	15	0.86	0	1000	548
6	1011#00	D4 D2 d D4 D2 d &	15	0.99	0	0	611
7	00#1011	D2 ∼ ∼	15	0.99	0	1000	2163
8	010#100	D3	15	0.94	0	0	2828
9	11010#1	D5 D5 & D5 D5 & d	14	0.88	0	1000	672
10	101010#	D0 D6 d	14	0.98	0	0	1142
11	00#0000	D2	14	0.99	0	0	3537
12	111110#	D6 D6 d	13	0.98	0	0	972
13	00#0001	D1 D2 | D1 D2 | |	13	1	0	1000	1400
14	00#0010	D2 D6 r	13	0.84	0	0	1715
15	0#01110	D4 D1 D1 d &	13	0.96	0	0	1818

overlapping natured problems. These two properties will be experimentally verified in the future. In the work presented here, only Boolean classification problems were tested. The XCSCFA systems will be extended and tested for multi-step problems and the problems involving more than two actions.

8. REFERENCES

[1] M. Ahluwalia and L. Bull. A Genetic Programming Based Classifier System. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 11–18, 1999.

[2] L. Bull and T. O'Hara. Accuracy-based Neuro And Neuro-Fuzzy Classifier Systems. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 905–911, 2002.

[3] M. V. Butz, T. Kovacs, P. L. Lanzi, and S. W. Wilson. Toward a Theory of Generalization and Learning in XCS. *IEEE Transactions on Evolutionary Computation*, 8(1):28–46, 2004.

[4] M. V. Butz, P. L. Lanzi, and S. W. Wilson. Function Approximation With XCS: Hyperellipsoidal Conditions, Recursive Least Squares, and Compaction. *IEEE Transactions on Evolutionary Computation*, 12(3):355–376, 2008.

[5] M. V. Butz, M. Pelikan, X. Llorà, and D. E. Goldberg. Automated Global Structure Extraction for Effective Local Building Block Processing in XCS. *Evolutionary Computation*, 14(3):345–380, 2006.

[6] M. V. Butz and S. W. Wilson. An Algorithmic Description of XCS. *Soft Computing*, 6(3-4):144–153, 2002.

[7] H. H. Dam, H. A. Abbass, C. Lokan, and X. Yao. Neural-Based Learning Classifier Systems. *IEEE Transactions on Knowledge and Data Engineering*, 20(1):26–39, 2008.

[8] S. P. Dandamudi. *Fundamentals of Computer Organization and Design*. Springer, 2003.

[9] C. Ioannides, G. Barrett, and K. Eder. Improving XCS Performance on Overlapping Binary Problems. In *Proceedings of the Congress on Evolutionary Computation*, pages 1420–1427, 2011.

[10] C. Ioannides, G. Barrett, and K. Eder. XCS Cannot Learn All Boolean Functions. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1283–1290, 2011.

[11] M. Iqbal, W. N. Browne, and M. Zhang. Extracting and Using Building Blocks of Knowledge in Learning Classifier Systems. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 863–870, 2012.

[12] M. Iqbal, W. N. Browne, and M. Zhang. XCSR with Computed Continuous Action. In *Proceedings of the Australasian Joint Conference on Artificial Intelligence*, pages 350–361, 2012.

[13] M. Iqbal, W. N. Browne, and M. Zhang. Evolving Optimum Populations with XCS Classifier Systems. *Soft Computing*, 17(3):503–518, 2013.

[14] M. Iqbal, W. N. Browne, and M. Zhang. Extending Learning Classifier System with Cyclic Graphs for Scalability on Complex, Large-Scale Boolean Problems. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 2013. to appear.

[15] M. Iqbal, W. N. Browne, and M. Zhang. Learning Overlapping Natured and Niche Imbalance Boolean Problems Using XCS Classifier Systems. In *Proceedings of the Congress on Evolutionary Computation*, 2013. to appear.

[16] M. Iqbal, W. N. Browne, and M. Zhang. Reusing Building Blocks of Extracted Knowledge to Solve Complex, Large-Scale Boolean Problems. *IEEE Transactions on Evolutionary Computation*, 2013. under review.

[17] M. Iqbal, M. Zhang, and W. N. Browne. Automatically Defined Functions for Learning Classifier Systems. In *Proceedings of the Genetic and Evolutionary Computation Conference (Companion)*, pages 375–382, 2011.

[18] T. Kovacs. Evolving Optimal Populations with XCS Classifier Systems. Technical Report CSR-96-17 and CSRP-9617, University of Birmingham, UK, 1996.

[19] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, 1992.

[20] P. L. Lanzi. XCS with Stack-Based Genetic Programming. In *Proceedings of the Congress on Evolutionary Computation*, pages 1186–1191, 2003.

[21] P. L. Lanzi and D. Loiacono. Classifier Systems that Compute Action Mappings. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1822–1829, 2007.

[22] D. Loiacono, A. Marelli, and P. Lanzi. Support Vector Machines for Computing Action Mappings in Learning Classifier Systems. In *Proceedings of the Congress on Evolutionary Computation*, pages 2141–2148, 2007.

[23] E. F. Moore. Gedanken-experiments on Sequential Machines. In *Automata Studies: Annals of Mathematics Studies*, pages 129–153. Princeton University Press, 1956.

[24] S. W. Wilson. Classifier Fitness Based on Accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.

[25] S. W. Wilson. Classifiers that Approximate Functions. *Natural Computing*, 1:211–233, 2002.